Slide 1

Administrivia

- Reminder: Homework 7 due today, Homework 8 Wednesday. Late penalties will apply if you can't turn them in on time, but will be reduced. (Many students seem to be busy?)
- Reminder: Quiz 6 Wednesday. Likely topic is pipelining. Probably high-level conceptual questions.

Slide 2

Minute Essay From Last Lecture

- Many interesting and thoughtful answers!

Slide 3

Pipelining — Key Ideas

- Instruction execution divided into stages.
- Full circuit partitioned into corresponding stages, with “pipeline registers” between them.
- All stages (can be) active at the same time, each operating on a different instruction.
- Each instruction moves through the pipeline.
- Performance improvement is in throughput rather than time for each instruction.
- “Hazards” complicate matters.

Slide 4

Pipelined Implementation — Some Details

- Figures 4.36 through 4.40 show some details of how this implementation works for different groups of instructions. Textbook’s notation is that state elements whose right side is highlighted (blue) are being read, and those whose left side is highlighted are being written.
- Note that we now spot a flaw in the design: At the point where we need “which register to write to?”, it’s no longer correct. Figure 4.41 shows how to correct.

Slide 5

Pipelined Implementation — What's Left

- Need to be explicit about exactly what's needed for those "registers" between stages, but should sort of be common sense(?).
- Need to generate control signals, as in single-cycle implementation. Note that some of them must be saved in those interstage registers. Figure 4.51 shows result.
- Need to deal with data and control hazards. (Structural hazards don't exist for MIPS ISA, assuming we have separate instruction/data memories, as in the single-cycle implementation.)

  Textbook shows many details, interesting but a bit much for this course. But good to get key ideas . . .

Slide 6

Data Hazards — Overview

- Some kinds of data hazards can be addressed by providing additional paths for data to flow ("forwarding"). For others, have to stall the pipeline.

  (Figures 4.53, 4.56.)
- "Stall the pipeline"? can get that effect by not changing registers or memory, and not changing program counter (so in effect the instruction being fetched is fetched again), and/or by inserting a `nop` instruction on the fly.
- Smart compilers can (at least sometimes) avoid stalls by reordering instructions.

Slide 7

Control Hazards — Overview

- Several ways to deal with control hazards:
- Could just stall pipeline. (Apparently not done.)
- Or could implement “delayed branches” — always execute instruction after the branch. (Look at figures and confirm that this will work.) Apparently what MIPS does? (So SPIM not quite accurate implementation of ISA.) Annoying if writing assembly-language programs, but few people do, and compilers can cope?
- Still other ways (used in other architectures?) involve “flushing” in-progress instructions (before they change anything!), possibly combined with various schemes for predicting branch outcome. Details no doubt interesting, but not trivial!

Slide 8

Exceptions

- As in higher-level programming languages, situations at this level where you want to bail out of the normal flow of control because something has gone wrong (e.g., arithmetic overflow).
- Further, situations in which you want to alter normal flow of control to deal with something happening outside processor (e.g., I/O device has finished something you previously asked it to do). (You could check it periodically, yes, but usually that’s inefficient.)
- Some architectures distinguish between “exceptions” (first case) and “interrupts” (second case), but all kind of the same thing, so MIPS doesn’t; all “exceptions”.
- What should happen on exception? Several possibilities . . .

Slide 9

Exceptions, Continued

- Some exceptions errors from which we can't reasonably recover (e.g., program tried to execute something not an instruction).

  What should happen then? probably terminate the offending program.
- Other exceptions errors from which recovery is possible, or things that have nothing to do with currently-running application (e.g., signal from I/O device).

  What should happen then? operating system should do something and then return to interrupted application.
- Exception/interrupt mechanism turns out to also be useful as a way for applications to request operating-system services.

Slide 10

Exceptions — Hardware Versus Software

- Hardware must save current PC (with a caveat) and transfer control to fixed location(s) with an indication of cause of exception.
- Code at fixed location(s) must "do the right thing" for the exception, as described previously. Normally this code is part of operating system.
- Caveat: Pipelining complicates exception processing — must allow instructions prior to the interrupted one to complete, complete or flush the interrupted one, etc. Textbook has (some of) details.

Hardware for Exceptions

Slide 11

- So, on exceptions (any type) need to bypass normal flow of control and branch to — somewhere, and fixed location(s) seems reasonable(?).
- Also need some way of indicating type of exception, plus address of interrupted instruction (in case we need to go back).

Hardware for Exceptions, Continued

Slide 12

- MIPS architecture uses two registers
  - cause of exception (“Cause register”)
  - address of interrupted instruction (EPC)

  and always transfers control to same place (where there should be code that’s part of operating system).

  (Compare Figures 4.65, 4.66.)

  (Try, in SPIM, a program that forces an exception — `sw` to an invalid address seems to work.)
- Other architectures transfer control to different places depending on type of exception — “vectored interrupts”.

Slide 13

Minute Essay

- None — quiz.